ReadMe:

- Apriori and FPGrowthTree are implemented as Apriori.py and FPGross.py

  The Main function for report is implemented as Project1.py. Simply run Project1.py will yield the timing information.

- Two algorithms are called with the following method:

Apriori:
FrequentPatternSet: Apriori.build_apriori(data:Array[][], Minsupport:Int =1)

FPgrowth:
FrequentPatternSet: FPgross(data:Array[][], Minsupport:Int =1).minePatters(Minsupport:Int)

- For reasons analyzed below, running full adult data with Apriori is expensive. The main comments out this operation, replicate result at own risks.

Report:

Part1: Running time comparison:

Setting min_support to be 0.5,

Test1, using the 10*5 dataset in textbook illustrated:
Apriori: 14.7459506989 miller Seconds on average.
FPgrowth: 2.8989315033 miller Seconds on average.

Test2, using pruned adultData (100rows):
Apriori: 50.2550601959 Miller seconds on average.
FPgrowth: 15.6199932098 Miller seconds on average.

Test3, using full adultData:
Apriori: More than 2 hours.
FPgrowth: 2,629.18996811 miller seconds on average.

In conclusion,
FPgrowth is at least of a magnitude faster than Apriori. Especially when the data gets larger. Test3 is a great illustration of the case where Apriori runs more than 2 hours while FPgrowth takes care of it in 3 seconds.

However, the partitioning attribute, such that Apriori can be implemented locally on each partition allows the potential of parallelism. Giving a big number of nodes Apriori might be as good as(if not better than) FPGrowth.

Unfortunately the nature of FPGrowth prohibits parallelism. That is to say, it is almost impossible to implement such algorithm on hdfs-based big data. Because traversing the dataset twice with a single process is technically impossible.

Part2: improvement of Apriori

- Apart from the partitioning mentioned in the previous part, another feasible improvement of Apriori is transaction removal.
  That is, when computing Lk-1, if a row fails to contribute to any of the potential Lk-1 frequent items set, it is removed from the Lk consideration.

- My Apriori.py adopted transaction removal improvement. However further tests suggest that this improvement does not accelerate the algorithm at a significant level. Even, the improvement slows the algorithm in larger datasets.
  Further analysis suggests that "removing the transaction from the dataset" from the textbook should not be interpreted as calling **data.remove(transaction).** The right way to implement is calling **data.Flag(transaction)** in order to exclude further scanning. Because remove in most of the data structure is called with O(n). This is a fun fact to be noticed.

- Also, because the heaviest amount of computation is concentrated in the first several iterations where small-sized item set are of high volume. During which stage, almost most of the transactions have certain contribution toward the support count. Transactions that can be removed from the data is of limited amount. Thus the percentage improvement is upper-bounded by a certain number that is not very impressive.

- Also, although implemented to support panda.Dataframe, Apriori.py is very costive under the panda.dataframe. Experiments carried outside of the project shows that if data is constructed with Array[][], Running time is greatly reduced by a magnitude.

- Continuous variables in the data will cause longer running time, Binning or remove continuous variables will improve results.